Chapter 10 A Prototype COncurrent Designer's Assistant

In the preceding seven chapters of this dissertation, a knowledge-based method is proposed and then specified to provide software designers with automated assistance when concurrent designs must be generated from data/control flow diagrams. In addition to proposing and specifying this knowledge-based method, the research associated with this dissertation includes a prototype COconcurrent Designer's Assistant, briefly, CODA, in order to achieve the following objectives:

- To assess the feasibility of the proposed approach,
- To evaluate the specifications for the meta-models, the design rules, and the design meta-knowledge, and
- To determine strengths and weaknesses of the proposed approach.

To address these objectives, CODA is applied to four concurrent-design problems, as reported in Appendices B, C, D, and E. The main purpose of the current chapter is to describe CODA. The description addresses four topics. First, CODA's software architecture is described and related to the conceptual architecture introduced and discussed previously in Chapter 3. Second, the knowledge representation techniques used to implement CODA are explained. Third, the components of CODA's software architecture are described. Fourth, CODA is presented from the user's viewpoint. Two types of users are considered: the novice designer and the experienced designer.

10.1 Software Architecture for CODA

The prototype, CODA, embodies a software architecture, as shown in Figure 29. In the figure, rectangles represent components of CODA's software architecture; disk symbols represent external repositories of information and knowledge; data store symbols depict loaded instances of the various meta-models; solid, directed arcs indicate flows of data among CODA's components, data repositories, and computer memory; dashed, directed arcs denote the flow of control without associated data; combined, dashed and solid, directed arcs signify the flow of control with associated data.

The components of CODA's software architecture differ from the elements of the conceptual architecture proposed in Chapter 3 (see Figure 1). These differences reflect packaging decisions made when translating elements from the conceptual architecture into implementation constructs supported by a specific expert system shell, CLIPS, Version 6.0. [NASA93] The components of CODA's software architecture can be mapped, however, to and from elements of the conceptual architecture. Table 6 provides such a mapping.

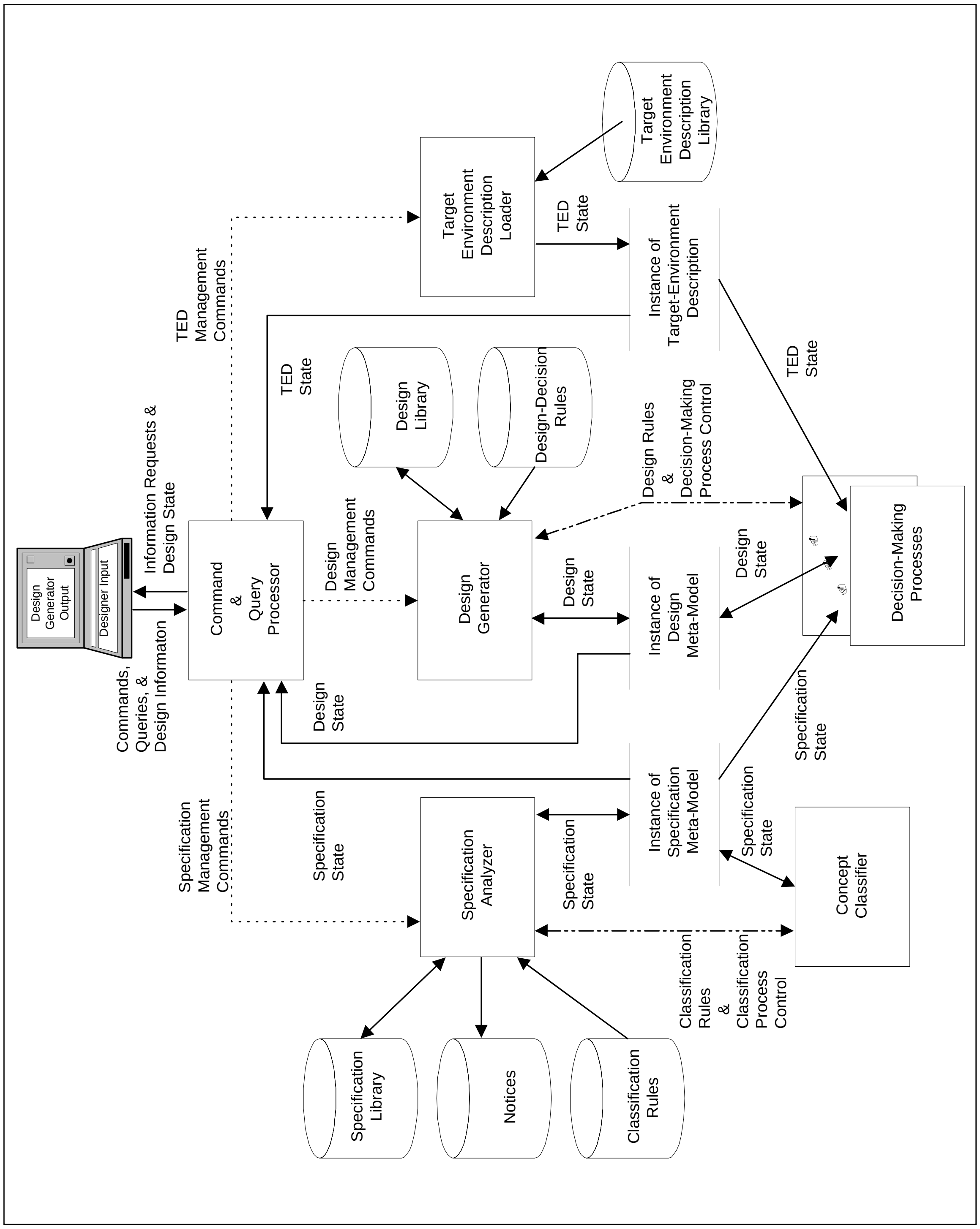
Design Generator Output
Designer Input
Commands, Queries, & Design Informaton
Information Requests & Design State
Command & Query Processor
TED Management Commands
Target Environment Description Loader
Target Environment Description Library
TED State
Instance of Target-Environment Description
TED State
TED State
Design Management Commands
Design Library
Design-Decision Rules
Design Generator
Design Rules & Decision-Making Process Control
Design State
Instance of Design Meta-Model
Design State
Decision-Making Processes
Design State
Specification State
Specification Management Commands
Specification State
Instance of Specification Meta-Model
Specification Analyzer
Specification State
Specification State
Concept Classifier
Classification Rules & Classification Process Control
Specification Library
Notices
Classification Rules

Figure 29. Software Architecture for Prototype COconcurrent Designer’s Assistant

Table 6. Mapping Conceptual Architecture to Prototype Software Architecture

Conceptual Architecture Component	Prototype Software Component(s)
Design Process Meta-Knowledge	Command & Query Processor Specification Analyzer Design Generator
User Interface Meta-Knowledge	Command & Query Processor
Specification Analysis, Inference, and Elicitation Knowledge	Specification Analyzer Concept Classifier Classification Rules
Design Generation Knowledge	Design Generator Decision-Making Processes Design-Decision Rules
Specification Meta-Model	Instance of Specification Meta-Model
Design Meta-Model	Instance of Design Meta-Model
Target Environment Description Meta-Model	Instance of Target Environment Description
User Interface Knowledge	Command & Query Processor
Target Environment Elicitation Knowledge	Off-Line Editor
Specification Model Instances	Specification Library
Design Model Instances	Design Library
Target Environment Description Model Instances	Target Environment Description Library

CODA distributes Design Process and User Interface Meta-knowledge among three software components: the Command & Query Processor, the Specification Analyzer, and the Design Generator. CODA implements User-Interface Knowledge mainly in the Command & Query Processor. CODA splits the Specification Analysis, Inference, and Elicitation Knowledge among three software components: the

Specification Analyzer, the Concept Classifier, and the Classification Rules. Similarly, CODA distributes Design-Generation Knowledge among the Design Generator, the Decision-Making Processes, and the Design-Decision Rules. CODA implements the Specification Meta-Model, the Design Meta-Model, and the Target Environment Description Meta-Model as meta-data for describing instances of specifications, designs, and target environment descriptions. Such instances may be loaded into CODA's memory, but otherwise reside in the appropriate library. CODA does not directly support the Target Environment Description Elicitation Knowledge. Instead, the designer can create and store target environment descriptions off-line, using a text editor.

10.2 Knowledge Representation

The prototype CODA is implemented using an expert system shell, the C Language Integrated Production System, or CLIPS, Version 6.0. [NASA93] CLIPS provides numerous techniques for representing knowledge, including:

- production rules for representing heuristic knowledge,
- an object-oriented query language for representing constraints and axiomatic knowledge,
- specific and generic functions and a procedural programming language for representing control knowledge,
- modules for bounding the scope of knowledge,
- a complete object-oriented language, and

- several forms for representing facts, including unstructured and structured facts and object-oriented data models.

Most of these knowledge representation techniques are used in the prototype to implement aspects of the knowledge specified in Chapters 4 through 9.

10.2.1 Specification Meta-Model

Consider first the knowledge representation implemented for the Specification Meta-Model, described in Chapter 4 and Appendix A. Table 7 presents the essential mappings.

Table 7. Knowledge Representation for the Specification Meta-Model

Specification Meta-Model Component	**Knowledge-based Representation**
Concept	Object Class
Concept Hierarchy	Class-Inheritance Hierarchy
Concept Axiom	Class-Query Specification
Classification Checking	Class Method and Subclass Checking
Axiom Checking	Class Method and Class-Query Evaluation
Concept Classification	Class-based Expert-System Rules
Information Elicitation	Procedure within a Function

CODA represents the Specification Meta-Model as a semantic data model containing two major components: concepts and concept relationships. Each concept is represented as an object class. Two types of concept relationships exist. CODA expresses one type, classification relationships, through a class-inheritance hierarchy. CODA expresses the second type, axiomatic relationships, as concept axioms, where each axiom is represented with a specification for a class query.

The Specification Meta-Model facilitates four forms of analysis, where each form is implemented using knowledge representation techniques available in CLIPS, Version 6.0. One form of analysis, classification checking, enables an instance of a concept to determine if it is a leaf within the concept hierarchy. Classification checking is implemented using a class method, known in CLIPS as a message handler, assigned to the object class that represents the concept Specification Element. This method is inherited by any concept in the concept hierarchy that inherits the concept Specification Element. The method simply determines whether or not the instance of the concept has any subclasses. If not, then the concept is a leaf within the concept hierarchy. A second form of analysis, axiom checking, enables an instance of a concept to determine if all axioms appropriate to the concept are satisfied. Axiom checking is implemented using a class method assigned to each concept. The class method evaluates each query specification that represents an axiom applicable to the concept. If all applicable query specifications are satisfied for a given instance of a concept, and for all of its parents in the concept hierarchy, then the axiomatic relationships described for that concept instance are valid. A query specification is implemented for each of the axioms specified in Appendix A.1. A third form of analysis, concept classification, enables a concept instance to be classified more specifically within the concept hierarchy. Concept classification is implemented using class-based, expert-system rules. One rule is defined for each situation where a more specific concept can be inferred to replace an existing but more general concept. An expert-system rule is implemented for each of the classification rules specified in Appendix A.2. The classification rules are packaged into a four-stage, inference network that composes the Concept Classifier described in Chapter 4. A fourth form of analysis, information elicitation, determines where additional information is required or desired in order to make subsequent design decisions. The designer is forced to supply any missing but required information. The designer is also shown where additional information might be desirable and is given an opportunity to supply it. Elicitation of each category of missing information is represented as a function containing a procedural algorithm within it. The categories of elicitation include: timer periods, maximum rates, exclusion groups, aggregation groups, locked-state events, and cardinalities.

10.2.2 Design Meta-Model

Another major component of CODA provides a means for representing concurrent designs and target-environment descriptions. CODA implements the Design Meta-Model specified in Chapter 5. Table 8 shows the essential mappings between elements of the Design Meta-Model and corresponding knowledge-based representations used in CODA.

The design meta-model is defined via an entity-relationship, or E-R, diagram containing two major components: design entities and relationships. CODA represents each design entity as an object class. Two types of relationships exist. CODA represents one type, inheritance relationships, through a class-inheritance hierarchy. CODA expresses each instance of the second type, arbitrary design relationships in the E-R diagram, as an object class.

Table 8. Knowledge Representation for the Design Meta-Model

Design Meta-Model Component	**Knowledge-based Representation**
Design Entity	Object Class
Design-Entity Hierarchy	Class-Inheritance Hierarchy
Design Relationship	Object Class
Design Constraint	Class Method and Class-Query Specification
Design Guideline	Class Attribute
Target-Environment Characteristic	Class Attribute

Where a design relationship requires constraints those constraints are represented using two approaches. Constraints restricting the types of entities that may be involved within a design relationship are represented with class methods, where each design relationship can have a class method defined that enforces the applicable constraints. Constraints describing a consistent and complete design are represented using class-query specifications that are contained within a function. To determine whether or not a given instance of a design meets the consistency and completeness constraints, the specified queries can be evaluated using a call to the appropriate function.

The remaining components of the Design Meta-Model describe characteristics of target environments and design guidelines. A single object class represents the collection of relevant characteristics and guidelines. Each characteristic and each guideline is represented as a class attribute within the object class.

10.2.3 Decision-Making Processes and Design-Decision Rules

Each of the Chapters 6 through 9 identifies a series of decision-making processes that compose a specific phase in the design-generation process. For each decision-making process a set of design-decision rules is specified. CODA contains knowledge representations for both decision-making processes and for design-decision rules. Each decision-making process is represented as a CLIPS module. A CLIPS module can be used to bound the scope of rule sets tagged as belonging to the module. CLIPS then allows the focus of decision-making activity to be moved from module to module. Rules defined as belonging to a module are applicable only when the focus of decision-making activity is the module to which the rules belong. The design-decision rules themselves are represented as CLIPS rules. A CLIPS rule exists for each rule specified within Chapters 6 through 9. Each rule is tagged as belonging to a module that corresponds to the decision-making process that contains the rule.

10.2.4 Meta-Knowledge

CODA represents meta-knowledge, which controls the analysis of specifications and the generation of designs, in several different forms, depending upon the purpose of the meta-knowledge. The discussion that follows refers to Table 9.

Table 9. Representations for Meta-Knowledge

Meta-Knowledge Component	**Knowledge-based Representation**
Specification Analyzer	Object Class
Specification-Analysis Phase	Class Method and External Function
Ordering Specification-Analysis Phases	Procedure within an External Function
Design Generator	Object Class
Design-Generation Phase	Class Method and External Function
Ordering Design-Generation Phases	Procedure within an External Function
Ordering Decision-making Processes	Procedure within a Class Method

Each of the two major activities, specification analysis and design generation, required to transform an input specification into a concurrent design, is represented as an object class, the Specification Analyzer and the Design Generator, respectively. Each phase of specification analysis, that is, conditioning the specification, checking classifications, and checking axioms, is represented with a class method in the Specification Analyzer. In addition, an external function is also used to represent each phase of specification analysis. Each external function contains procedural knowledge that prevents improper invocation of the specification-analysis phase that is represented by the function. In effect, each external function checks the ordering constraints for its corresponding class method in the Specification Analyzer. Each phase in the process is represented redundantly with both an external function and a class method in order to provide a convenient means for an experienced designer to invoke a process phase. The syntax for invoking CLIPS functions is more succinct than the syntax for invoking methods within CLIPS object classes. The same approach is used to represent the meta-knowledge that constrains the order of the phases composing the design generation process, that is, structuring tasks, defining task interfaces, structuring modules, and integrating tasks and modules. The remaining form of meta-knowledge orders the invocation of decision-making processes within each phase of design generation. The ordering of decision-making processes is represented as procedural knowledge encoded within each class method that corresponds to a design-generation phase.

10.3 CODA's Components

The next few sections provide a description of the responsibilities held by the key components in CODA's software architecture. Refer back to Figure 29 for a diagrammatic view of the relationships among the various components.

10.3.1 Command & Query Processor

The Command & Query Processor provides the means for a designer to interact with CODA. An experienced designer can issue either commands, which can cause the state of the CODA to change, or queries, which can report on, but not change, the state of the CODA. The Command & Query Processor ensures that appropriate ordering constraints are satisfied before passing each command on to the appropriate component, that is, the Specification Analyzer, the Design Generator, or the Target Environment Description Loader, for execution. In general, the Command & Query Processor executes queries by examining directly the in-memory state of a specification instance, a design instance, and a target environment description instance, where only one instance of each type can exist in memory at a given time.

10.3.2 Specification Library and Specification Analyzer

The Specification Library, created using a process external to CODA, provides a repository of specification instances, where each instance represents a data/control flow diagram, augmented with any additional information. At present, specification instances are created and added to the library using a text editor. A program could be created to generate specifications by extracting relevant data from repositories associated with computer-aided software engineering (CASE) tools that enable data/control flow diagrams to be represented. Specifications within the library are updated as a result of specification analysis.

The Specification Analyzer executes commands to move specification instances between computer memory and the Specification Library, to check properties of loaded specifications, to load classification rules, and to control the process for classifying specification concepts. When a specification is analyzed, each concept classification, axiom violation, and classification deficiency, as well as, any information elicited from the user, is logged to a file, Notices.

10.3.3 Concept Classifier and Classification Rules

The Concept Classifier, when asked by the Specification Analyzer, attempts to classify concepts for a loaded specification in accordance with the Specification Meta-Model described in Chapter 4. The Classification Rules, encoded as expert-system rules, are included as an integral part of CODA.

10.3.4 Design Library and Design Generator

The Design Library, updated by CODA as designs are generated, provides a repository of designs hierarchically organized by the specification from which a design originated and then by the target environment for which that design is intended. The Design Library, then, provides a tree structure for locating generated designs.

The Design Generator executes commands to move design instances between computer memory and the Design Library, to initiate each design phase, to manage the decision-making processes that compose each design phase, and to load the design-decision rules associated with each decision-making process. Each decision-making process executes a set of design-decision rules that update the state of the design as required, based on an examination of the current specification, the target environment description, and the evolving design. The Design-Decision Rules, encoded as expert-system rules, are included as an integral part of CODA.

10.3.5 Target Environment Description Library and Loader

The Target Environment Description Library provides a repository of target environment descriptions known to CODA. With the current prototype CODA, Target Environment Descriptions are created and added to the library using a text editor. The prototype could be extended easily to allow Target Environment Descriptions to be created and added to the library without resorting to an external editor. The Target Environment Description Loader executes a command that loads a requested Target Environment Description from the library.

10.3.6 Hardware and Software Requirements for the Prototype

CODA is implemented to execute under Windows 3.1 and under UNIX, but requires access to a run-time environment for CLIPS, Version 6.0, or later The prototype is known to execute on an Intel 486 computer system and on a Sun Sparcstation 2.

10.4 User's View of CODA

The current prototype CODA relies upon a rather crude text dialog interface with the designer. CODA's power could be better exploited by providing a graphical user interface that works directly on pictures of data/control flow diagrams and concurrent designs. The startup dialog for CODA is shown in Figure 30. After an explanation of the purpose of CODA, the designer is given a chance to exit. If the designer opts to continue, then CODA asks about the designer's level of experience because some design decisions are taken only after consultation when an experienced designer is available, but are taken without consultation when an experienced designer is unavailable. In addition, the manner in which CODA proceeds to generate designs depends upon whether or not the designer professes to be experienced.

10.4.1 CODA Viewed by a Novice Designer

Should the designer profess to be inexperienced, CODA leads the designer through the steps necessary to analyze a data/control flow diagram and, subsequently, to generate a concurrent design. First, the designer is shown the data/control flow diagrams that exist in CODA's Specification Library. The designer must select one of those specifications. Next, the designer is shown the list of target environment descriptions known to CODA and is asked to select one. Once the input specification and the target environment are known, CODA attempts to classify the concepts in the specification and then elicits any additional information that is required from the designer. The designer is also given opportunities to provide any additional information that can help CODA improve its decision-making. After classifying concepts in the input specification, CODA checks each concept to ensure it is fully classified and that all applicable axioms are satisfied. Failing this, the designer is asked to repair the errors in the specification

```
         WELCOME TO CODA

The COncurrent Designer’s Assistant, or CODA, is
an experimental program that embodies knowledge
about generating concurrent designs from input
specifications in the form of data/control flow
diagrams.  This design knowledge is based on a
restricted form of RTSA, known as COBRA, and on
a design method known as CODARTS.  The knowledge
within CODA is represented using Semantic Data
Modeling and Rule-based Expert Systems.

Do you wish to use CODA?

   1.  Yes
   2.  No

Please select a number?
1

CODA consults experienced designers where
necessary to make certain design judgments.
In addition, CODA relies upon experienced
designers to direct the design process.  For
inexperienced designers, CODA takes certain
design decisions by default.  In addition,
CODA leads inexperienced designers through
the design process.

What is your level of experience at designing
concurrent software?

   1.  Experienced Designer
   2.  Inexperienced Designer

Please select a number? 1

The design generator operates through a set of
commands and queries.  A command initiates some
operation within the design generator.  A query
lists some aspect of the current input specification,
target environment description, or evolving design.

Each command and query must be enclosed in parentheses.

(commands) -- lists the available commands
```

Figure 30. CODA Startup Dialog

and CODA terminates. If the specification analysis succeeds, then the designer is asked whether to save the updated specification before proceeding with design generation.

Given a proper specification, CODA next initiates design generation. Tasks are structured and then task interfaces are defined. Next, modules are structured and the task and module views of the design are integrated. Once the design exists, CODA writes task behavior specifications, module specifications, and task and module design histories into the design library. Next, CODA checks the generated design for completeness and consistency, recording the results in a design summary. Finally, the designer is given an opportunity to save the updated specification and generated design into the Specification Library and Design Library, respectively.

10.4.2 CODA Viewed by an Experienced Designer

An experienced designer might prefer to generate and review designs a little at a time. In this way, intermediate steps can be discarded should the designer be dissatisfied with the results. Alternatively, the designer might wish to save a partial design for reuse later, perhaps when the design must be moved to an alternate target environment. For these reasons, and because an experienced designer understands the design process, CODA relies upon an experienced designer to access commands and queries interactively on demand. At startup, CODA shows the experienced designer how to access the list of commands and queries.

10.4.2.1 CODA Commands

The list of commands provided by CODA is shown in Figure 31. Each command listed in Figure 31 is implemented as a CLIPS function. Checking for ordering constraints is built into each command so that the command will not execute unless the appropriate constraints are satisfied. Violations of ordering constraints are referred to the designer. Corrective action usually requires invoking another command. An example is illustrated in Figure 32.

10.4.2.1.1 CODA Enforces Process Constraints

In the example, the **state** command is executed to reveal that no specification or target environment description is loaded and that no design is in progress. Subsequently, an attempt is made to execute three commands, **check-axioms**, **check-classes**, and **condition-spec**, each of which analyze some aspect of an input specification. As shown in Figure 32, each command is rebuffed because no specification is loaded. To correct the problem, the designer needs to load a specification before beginning the analysis. Similar checking is built into each command as necessary to represent the Design-Process Meta-Knowledge. When the ordering constraints for a command are satisfied, the command is referred to the appropriate CODA component, that is, the Specification Analyzer, Design Generator, or Target Environment Description Loader, for execution.

```
CLIPS> (commands)

(check-axioms)     : Checks axioms for loaded specification
(check-classes)    : Checks classifications for loaded specification
(checkpoint)        : Saves current state of design and specification
(condition-spec)    : Classifies loaded specification and then
                      elicits needed and desired information
(configure-design)  : Configures design for the target environment
(define-task-interfaces) : Generates interfaces for tasks in the design
(evaluate-design)    : Checks the design against axioms
(forget-design)      : Delete the current design from memory
(forget-spec)        : Delete the current specification from memory
(forget-ted)         : Delete the current target environment from memory
(integrate)          : Integrates the task and module views of the design
(list-specs)         : Lists the known specifications
(list-teds)          : Lists the known target environments
(load-spec ?name)    : Loads the named specification, ERASES DESIGN
(load-ted ?name)     : Loads the named target environment, ERASES DESIGN
(restore-design)     : Loads the last saved state of the current design
(save-spec)          : Saves the current specification
(state)              : Reports the current state of the design generator
(structure-ihms)     : Generates a module structure
(structure-tasks)    : Generates a task structure
(write-design)       : Writes task behavior specs., module specs., and
                           task and module design histories to disk.
```

Figure 31. CODA Commands Available to an Experienced Designer

```
CLIPS> (state)

Specification Loaded: None
  Conditioning:  Unknown
  Classification:  Unchecked
  Axioms:         Unevaluated

TED Loaded:  None

No Design In Progress

CLIPS> (check-axioms)
No Specification Loaded
CLIPS> (check-classes)
No Specification Loaded
CLIPS> (condition-spec)
No Specification Loaded
CLIPS>
```

Figure 32. Attempt to Analyze a Specification Before a Specification is Loaded

10.4.2.1.2 CODA Manages Libraries

A number of commands help to move elements between the various libraries and computer memory. The **list-specs** and **list-teds** commands provide a list of the contents of the Specification Library and Target Environment Description Library, respectively. The **load-spec** and **load-ted** commands move a named element from the appropriate library into memory. The **restore-design** command moves any existing design for the currently loaded specification and target environment description from the Design

Library into memory. The **state** command lists the status of any loaded specification, target environment description, and design. Three commands, **forget-spec**, **forget-ted**, and **forget-design**, erase any currently loaded specification, target environment description, and design, respectively, from computer memory. The **save-spec** command copies the currently loaded specification to the Specification Library. The **checkpoint** command copies both the currently loaded specification and the evolving design to the Specification Library and the Design Library, respectively.

10.4.2.1.3 CODA Analyzes Specifications

A second group of commands control the analysis, classification, and elicitation associated with an input specification. The **check-classes** command determines whether all concepts in an input specification are classified fully. Concepts lacking full classification are recorded via entries in a file named Notices. The **check-axioms** command determines whether or not each concept within an input specification satisfies the relevant axioms for a concept of its declared type. Whenever a concept axiom is violated then a notification is written to the Notices file. The **condition-spec** command attempts to classify completely, in accordance with the Specification Meta-Model, all concepts in an input specification. As each concept is classified, a notification is written to the Notices file. In addition, the command identifies and elicits from the designer any additional information necessary to make design decisions, or desired to improve the quality of design decisions.

10.4.2.1.4 CODA Generates Designs

The remaining commands embody Design-Process Meta-Knowledge, described in Chapter 3, and the Design-Generation Knowledge, discussed in detail in Chapters 6, 7, 8, and 9. After checking that the appropriate ordering constraints are satisfied, the **structure-tasks** command invokes the Task Structuring Knowledge base (see Chapter 6). Similarly, the **define-task-interfaces** command ensures that a task structure exists and then invokes the Task Interface Definition Knowledge base (see Chapter 7). The **structure-modules** command, after enforcing required ordering constraints, turns control over to the Module Structuring Knowledge base (see Chapter 8). The **integrate** command, after ensuring that both a task and module structure exists, invokes the Task and Module Integration Knowledge base (see Chapter 9). Another command, **write-design**, generates task behavior specifications, module specifications, and task and module design histories. These textual representations are written to the Design Library. In addition, **write-design** checks the generated design for completeness, relative to the input specification, and for consistency, relative to the Design Meta-Model. The results of this completeness and consistency checking, along with an index of the tasks and modules generated for the design, are written, as a design summary, into the Design Library.

10.4.2.1.5 Unimplemented Commands

The two remaining commands, **configure-design** and **evaluate-design**, provide placeholders for additional phases in the design-generation process. The configuration and evaluation of designs is outside the scope of the research described in this dissertation.

10.4.2.2 CODA Queries

In addition to commands for analyzing specifications and generating designs, CODA provides the experienced designer with a set of canned queries that can reveal interesting details about the state of the input specification and the evolving design. Figure 33 lists the queries provided by the prototype CODA. Additional queries can easily be defined, tested, and included in the available set.

10.4.2.2.1 General Information Queries

A few queries examine the input specification or provide general information. Each entity in an input specification or a design is identified through a unique object identifier. When given a valid object identifier, the **tell** query reveals the class and name of an object, while the **print** query lists the class, the slots, and the slot values of an object. When given a valid specification-element type, the **any** query lists all instances of elements of that type within an input specification, while the **count** query reports the number of instances of elements of that type within an input specification.

10.4.2.2.2 Querying Design Elements

Most of the queries are aimed at the evolving design or at revealing connections between the input specification and the evolving design. The queries **tasks**, **ihms**, **data**, **events**, and **messages** print a list of the elements of the appropriate type in the evolving design. The query **no-tasks** lists all transformations in an input specification that are not

```
CLIPS> (queries)

(any ?type)         : Specification elements of type
(count ?type)       : Counts specification elements of type
(data)              : List of data inputs/outputs in design
(events)            : List of events in the design
(explain ?de)       : Rationale for design element
(ihms)              : IHMs in design
(inverses ?de)      : Inverse relationships for design element
(invert ?de ?type)  : Invert design relationships of type
                        for design element
(links ?de)         : Relationships for design element
(list ?de ?type)    : Relationships of type for design element
(messages)          : List of messages in the design
(no-ihm)            : Elements unallocated to an IHM
(no-placement)      : IHMs unplaced relative to the tasks
(no-tasks)          : Elements unallocated to a task
(print ?obj)        : Print slots for object
(tasks)             : Tasks in design
(tell ?obj)         : Class and Name for object
(traces-from ?de)   : Specification elements leading to
                        the input design element
(traces-to ?se)     : Design elements resulting from
                        the input specification element
(unallocated)       : Specification elements not allocated to
                        the design
```

Figure 33. CODA Queries Provided for Experienced Designers

allocated to a task. Similarly, the query **no-ihm** lists all transformations and data stores in an input specification that are not allocated to a module. The query **no-placement** identifies modules in a design that have not been placed relative to the tasks in a design. The query **unallocated** lists every specification element that is not allocated in the evolving design.

10.4.2.2.3 Querying Design Histories

When given the object identifier of a design element, the query **explain** prints the design history for the design element. A design history consists of a list of decision records, where each decision record includes the rule that made the decision, the action taken, and the rationale for the decision.

10.4.2.2.4 Querying Design Relationships

A number of queries reveal relationships among elements in a design. In effect, these queries allow a designer to navigate the relationships defined in the E-R model for concurrent designs. Given a valid object identifier for a design element, the **links** query lists all design relationships that involve the design element as either a subject or object. For example, a design element, Message, as an object, might be sent and received by two different tasks and, as a subject, might include a Message Data field. The **links** query would reveal all three of these relationships involving the Message. A more limited query, **inverses**, lists only the relationships in which a design element is the object. Using the previous example of the Message, the inverses query would reveal only the relationships that show that the message is sent and received by two tasks. To examine relationships of a selected type in which a design element is involved, the **list** query can be used. For example, **list oid Send**, would list all instances of the Send relationship involving the design element identified by the object identifier, **oid**. To reveal relationships of a selected type in which a design element is the object only, the **invert** query can be used.

The remaining queries reveal relationships between elements in the design and elements in the specification. When given the object identifier for a valid design element, the query **traces-from** lists all the specification elements that are allocated to the design element. Just the reverse, when given the object identifier for a valid specification element, the query **traces-to** lists all the design elements to which the specification element is allocated.